A Representation for Bit Section Based Analysis and Optimization*

Rajiv Gupta[1], Eduard Mehofer[2], and Youtao Zhang[1]

[1] Department of Computer Science, The University of Arizona, Tucson, Arizona
[2] Institute for Software Science, University of Vienna, Vienna, Austria

Abstract. Programs manipulating data at subword level are growing in number and importance. Examples are programs running on network processors, media processors, or general purpose processors with media extensions. In addition data compression techniques which are vital for embedded system applications result in code operating on subword level as well. Performing analysis on word level, however, is too coarse grain missing opportunities for optimizations. In this paper we introduce a novel program representation which allows reasoning at subword level. This is achieved by making accesses to subwords explicit. First in a local phase statements are analyzed and accesses at subword level identified. Then in a global phase the control-flow is taken into account and the accesses are related to one another. As a result various traditional analyses can be performed on our representation at subword level very easily. We discuss the algorithms for constructing the program representation in detail and illustrate their application with examples.

1 Introduction

Programs that manipulate data at subword level are growing in number and importance. The need to operate upon subword data arises if multiple data items are packed together into a single word of memory. The packing may be a characteristic of the application domain or it may be carried out automatically by the compiler. We have identified the following categories of applications.

Network processors are specialized processors that are being designed to efficiently manipulate packets [5]. Since a packet is a stream of bits the individual fields in the packet get mapped to subword entities within a memory location or may even be spread across multiple locations.

Media processors are special purpose processors to process media data (e.g., TigerSHARC [3]) as well as general purpose processors with multimedia extensions (e.g., Intel's MMX [1, 6]). The narrow width of media data is exploited by packing multiple data items in a single word and supporting instructions that are able to exploit subword parallelism.

* Supported by DARPA PAC/C Award. F29601-00-1-0183 and NSF grants CCR-0105355, CCR-0096122, EIA-9806525, and EIA-0080123 to the Univ. of Arizona.

Data compression transformations reduce the data memory footprint of the program [2,9]. After data compression transformations have been applied, the resulting code operates on subword entities.

Program analysis, which is the basis of optimization and code generation phases, is a challenging task for above programs since we need to reason about entities at subword level. Moreover, accesses at subword level are expressed in C (commonly used language in those application domains) by means of rather complex mask and shift operations.

In this paper we introduce a novel program representation that enables reasoning about subword entities corresponding to *bit sections* (a bit section is a sequence of consecutive bits within a word). This is made possible by explicitly expressing manipulation of bit sections and relating the flow of values among bit sections. We present algorithms for constructing this representation. The key steps in building our representation are as follows:

- By locally examining the bit operations in an expression appearing on the right hand side of an assignment, we identify the bit sections of interest. In particular, the word corresponding to the variable on the left hand side is split into a number of bit sections such that adjacent bit sections are modified differently by the assignment. The assignment statement is replaced by multiple bit section assignments.
- By carrying out global analysis, explicit relationships are established among different bit sections belonging to the same variable. These relationships are expressed by introducing split and combine nodes. A split node takes a larger bit section and replaces it by multiple smaller bit sections and a combine node takes multiple adjacent bit sections and replaces them by a single larger bit section.

The above representation is appropriate for reasoning about bit sections. For example, the flow of values among the bit sections can be easily traced in this representation resulting in definition-use chains at the bit section level. Moreover, since our representation makes accesses at subword level explicit, processors with special instructions for packet-level addressing can be supported easily and efficiently by the code generator and the costly mask and shift operations can be replaced.

The remainder of the paper is organized as follows. In section 2 we describe our representation including its form and its important properties. In sections 3 and 4 we present the local and global phases of the algorithm used to construct the representation. And finally concluding remarks are given in section 5.

2 The Representation

This section presents our representation for bit section based analyses and optimizations. Starting point for our extensions are programs modeled as directed control flow graphs (CFG) $G = (N, E, entry, exit)$ with node set N including the unique *entry* and *exit* nodes and edge set E. For the ease of presentation we assume that the nodes represent statements rather than basic blocks[1]. The construction of our representation is driven by assignment statements of the form $v = t$ whereby the right hand side term t contains bit operations only, i.e. & (and), | (or), *not* (not), << (shift left), and >> (shift right). Since the term on the right hand side of such an assignment can be arbitrarily long and intricate and since our goal is to replace those assignments by a sequence of simplified assignments, we call them *complex assignments.*

Essentially our representation is based on two transformations performed on the CFG. First we *partition* the original program variables into bit sections of interest. The bit sections of interest are identified *locally* by examining the usage of these bit sections in a complex assignment. Only complex assignments which are formed using bit operations are processed by this phase because partitioning is guided by the special semantics of bit operations. Other assignments are not partitioned since no useful additional information can be exposed in this way. Hence, in the remainder of the discussion, only complex assignments are considered. Second we *relate definitions and uses* of bit sections belonging to the same program variable using *global analysis.* The required program representation is obtained by making the outcomes of the above steps *explicit* in the CFG. In the remainder of this section, we illustrate the effects of the above two steps and describe the resulting representation in detail.

2.1 Identifying bit sections of interest

Definition 1 (Bit Section).
Given a program variable v with the size of c bits, a bit section of v is denoted by $v_{l..h}$ $(1 \leq l \leq h \leq c)$ and refers to the sequence of bits $l, l+1, .., h-1, h$.[2] *The symbol $:=$ is used to denote a bit section assignment.*

In the following discussion, if nothing is said to the contrary, we assume for the ease of discussion that variables have a size of 32 bits.

Partitioning a program variable. Given a *complex assignment* $(v = t)$, the program variable v on the left hand side is partitioned into bit sections, if each of the resulting sections is updated *differently* from its neighboring bit sections by the term t on the right hand side of the complex assignment. In particular, the value of a bit section of the lhs variable v, say $v_{l..h}$, can be specified in one of the following ways:

- *No Modification:* The value of $v_{l..h}$ remains unchanged because it is assigned its own value.
- *Constant Assignment:* $v_{l..h}$ is assigned a compile time constant.
- *Copy Assignment:* The value of another bit section variable is copied into $v_{l..h}$.
- *Expression Assignment:* The value of $v_{l..h}$ is determined by an expression which is in general simpler than t.

[1] Handling basic blocks is straightforward.
[2] The definition includes 1-bit sections as well as whole variable sections.

The partitioning of variable v is made explicit in the program representation by replacing the complex assignment by a series of *bit section assignments.* A consequence of this transformation is that operands used in t may also have to be partitioned into compatible bit sections.

Properties. There are two important properties that will be observed by our choice of *bit section partitions*:

1. *Non-overlapping sections.* The sections resulting from such partitioning are non-overlapping for individual assignments.
2. *Maximal sections.* Each section is as large as needed to expose the semantic information that can be extracted from a given complex assignment. In other words, further partitioning will not provide us with any more information about the values stored in the individual bits.

Example. Consider the complex assignment to variable a shown in Fig. 1. If we carefully examine this assignment, we observe that this complex assignment is equivalent to the bit section assignments shown below. Note that each bit section is updated differently from its neighboring sections. Bit sections $a_{1..4}$ and $a_{17..32}$ are set to 0, $a_{5..8}$ is involved in a copy assignment, and $a_{13..16}$ is not modified at all (we have placed the assignment below simply for clarity). Bit section $a_{9..12}$ is computed using an expression which is simpler than the original expression. Finally, as a consequence of a's partitioning, variable b must be partitioned into compatible bit sections as well.

<u>Complex Assignment</u>

$$a \;=\; (a \;\&\; 0xff00) \mid ((b \;\&\; 0xff) \;<<\; 4)$$

<u>Bit Section Assignments</u>

$a_{1..4}$	$:=$	0	`/* constant assignment */`
$a_{5..8}$	$:=$	$b_{1..4}$	`/* copy assignment */`
$a_{9..12}$	$:=$	$b_{5..8} \mid a_{9..12}$	`/* (simpler) expression */`
$a_{13..16}$	$:=$	$a_{13..16}$	`/* no modification */`
$a_{17..32}$	$:=$	0	`/* constant assignment */`

Fig. 1. Bit Section Assignments.

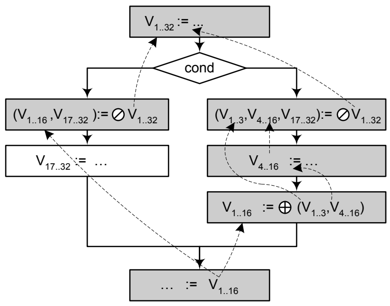
Note that bit section assignments reveal the information that some bit sections are set to constant values, others are assigned copies of bit sections, or some are unchanged. All this information would be lost if we reasoned about the variable a as a single 32 bit entity.

2.2 Establishing relationships among bit sections

After introducing bit sections for single complex assignments, the goal of this step is to establish relationships among the bit sections arising from different complex assignments. To illustrate the need for relating bit sections, let us consider the computation of definition-use relationships as shown in Fig. 2. The computation of definition-use relationships is complicated by the fact that left or right hand side occurrences of a given program variable may be partitioned differently by different complex assignments. In Fig. 2a partition $v_{17..32}$ and partition $v_{4..16}$ are created for variable v in the then-branch and else-branch, respectively. The conditional is followed by a use of partition $v_{1..16}$. However, the partitions do not match with each other and the relationships among the bit sections are hidden. Hence, we introduce special nodes in the program which *create* and *destroy* bit sections and make the relationships between bit sections explicit: A *split* node (denoted by $\oslash$) is introduced to *create* a set of smaller non-overlapping bit sections from a larger bit section and a *combine* node (denoted by $\oplus$) is introduced to *coalesce* smaller adjacent bit sections into a single longer bit section. The introduction of split and combine nodes as shown in Fig. 2b ensures that each section name exists before it is referenced. The edges show the flow of values to the use of $v_{1..16}$ at the end of the code fragment. By traversing these edges we can easily determine that if the then-branch is executed, the value reaching the use of $v_{1..16}$ at line 7 is defined by the assignment at line 1. On the other hand, if the else-branch is executed, the values of bits 1 through 3 defined at line 1 and values of bits 4 to 16 defined at line 5 reach the use of $v_{1..16}$ at line 7. Thus an important consequence of appropriately introducing split and combine nodes is that definition-use relationships can now be established among bit sections.

```
(1)     v1..32 := ...
(2)     if (cond) then
(3)         v17..32 := ...
(4)     else
(5)         v4..16 := ...
(6)     endif
(7)     ... := v1..16
```

(a) Bit section assignments.



(b) Def-use chains.

Fig. 2. Using Split and Combine Nodes.

Definition 2 (Split and Combine Nodes).
A **split node** *that partitions a bit section* $v_{l..h}$ *into* n *non-overlapping bit sections* $v_{l..s_1}, v_{s_1+1..s_2}, ..., v_{s_{n-1}+1..h}$ *is written as:*

$$(v_{l..s_1}, v_{s_1+1..s_2}, ..., v_{s_{n-1}+1..h}) \;:=\; \oslash v_{l..h}$$

Conversely a **combine node** *that merges adjacent bit sections* $v_{l..s_1}, v_{s_1+1..s_2}, ...,$ $v_{s_{n-1}+1..h}$ *into a single larger contiguous bit section* $v_{l..h}$ *is written as:*

$$v_{l..h} \;:=\; \oplus(v_{l..s_1}, v_{s_1+1..s_2}, ..., v_{s_{n-1}+1..h})$$

Properties. The rules for inserting split and combine nodes are derived from the following properties which shall hold for our representation.

1. *Non-overlapping sections.* While at different program points a variable may be partitioned differently, at each program point we associate a unique partitioning with a program variable into non-overlapping sections.
2. *Create-before-use.* Along all program paths, each right-hand side appearance of a bit section must be preceded by a left-hand side appearance of the same section.

The split and combine nodes act as transition points where non-overlapping partitioning of variable is changed from one partitioning to another. Given a program point where a split or combine node is placed, the node makes explicit the relationship among bit sections that existed immediately preceding the program point and following the program point.

Minimal representation. The properties described above which are essential for our representation can be realized by different placements of split and combine nodes. Of course, our goal is to meet the required properties with minimal insertions of split and combine nodes which leads to the following two *minimality criteria*:

1. The *number* of split and combine nodes introduced along some path shall be *minimal*. This can be achieved by ensuring that if a bit section is used repeatedly along a path, it is created once before its first reference and destroyed only after its last reference.
2. The *lifetime* of a bit section, i.e. the period during which the section name exists, shall be *minimal* (under the above criterion). This can be achieved by creating a section at the latest program point where it is needed and destroying it at the earliest program point where it is no longer needed.

The lifetime of a bit section starts and ends at a split or a combine node. More specifically, the appearance of a bit section on the left hand side of a split or combine node represents the *start point* of the bit section's lifetime. The appearance of a bit section on the right hand side of a split or combine node represents the *end point* of the bit section's lifetime.

The two minimality criteria mentioned above imply that the lifetimes of the bit sections are chosen such that they are long enough to reduce the need for split and combine nodes but not any longer. These criteria result in the following *placement strategy* for split and combine nodes.

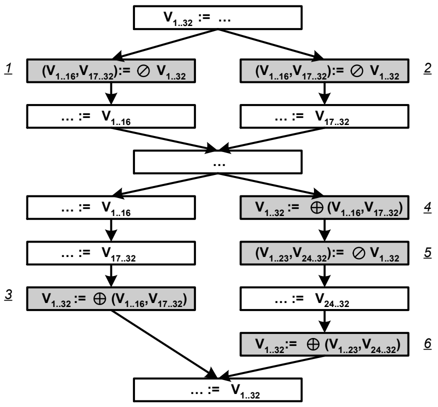
Earliest point placement of combine nodes. A combine node that destroys a bit section name is inserted at the *earliest program point* where the bit section is *not (partially) anticipable*, that is, it is known that an appearance of the bit section will no longer be encountered and therefore the bit section is no longer needed.

Latest point placement of split nodes. A split node that creates a bit section name is inserted at the *latest program point* at which the bit section is live but *not (partially) available*, that is, it does not already exist.

Example. Consider the code fragment of Fig. 3 together with the corresponding CFG with the inserted split and combine nodes displayed in dark boxes. Split nodes are placed at the *latest program points* immediately preceding the references to $v_{1..16}$, $v_{17..32}$, and $v_{24..32}$ at nodes 1, 2, and 5 in order to create those bit sections, since none of them are partially available. On the other hand, combine nodes are placed at the *earliest program points* just after the references to $v_{17..32}$ and $v_{24..32}$ at nodes 3 and 6 in order to destroy those bit sections since both are not used any more. Finally, since bit sections $v_{1..16}$ and $v_{17..32}$ are referenced in the then-branch of the second conditional statement but not in the else-branch, the *earliest program point* to destroy those bit sections is the very first statement of the else-branch at node 4.

```
v1..32 := ...
if () then
... := v1..16
else
... := v17..32
endif
if () then
... := v1..16
... := v17..32
else
... := v24..32
endif
... := v1..32
```

(a) Sample code.

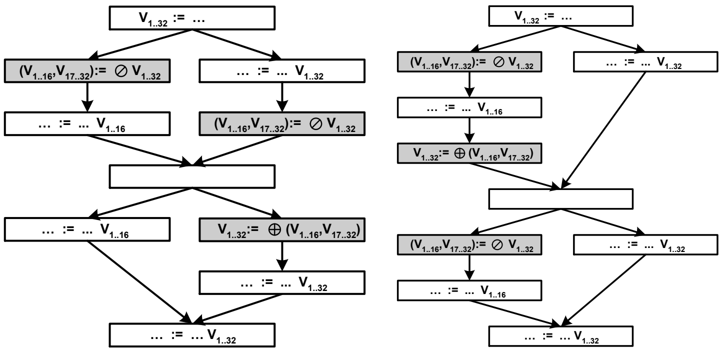


(b) Earliest/latest insertion of combines/splits.

Fig. 3. Placement of Split and Combine Nodes.

Minimal representation is not unique. In some situations multiple solutions are equally good under our criteria. As an example consider two consecutive if-statements with a use of $v_{1..16}$ in the then-branch and a use of $v_{1..32}$ in the else-branch as shown in Fig. 4a. If we decide to preserve bit section $v_{1..16}$ at the end of the first if-statement, we get solution Fig. 4b. On the other hand, if we decide to preserve bit section $v_{1..32}$, we get solution Fig. 4c. In both solutions three nodes are inserted, however, taking the left branches of the if-statements we have one inserted node in the first solution and three nodes in the second solution. On the other hand, taking the right branches we have two inserted nodes in the first solution but none in the second one. However, we consider either choice as equally good since both sections $v_{1..16}$ and $v_{1..32}$ have at the end of the first if-statement future references and both result in equal number of split and combine nodes. Note that our algorithm yields solution Fig. 4b.

```
v1..32 := ...
if () then
... := v1..16
else
... := v1..32
endif
if () then
... := v1..16
else
... := v1..32
endif
... := v1..32
```



(a) Source code. (b) Preserving $v_{1..16}$. (c) Preserving $v_{1..32}$.

Fig. 4. Multiple Minimal Solutions.

3 Local Phase: Identifying Relevant Bit Sections

In this phase the partitioning of left hand side (lhs) variables of complex assignments is determined under the constraint that each adjacent bit section shall be computed differently. This is done in two steps: First a *bottom-up* traversal of the right hand side (rhs) expression is carried out during which the *bit sections* required for the lhs variable are determined. Second the *bit section assignments* are generated in a *top-down* traversal of the rhs expression tree.

I. Finding bit sections of the lhs variable. The rhs expression tree is traversed in a bottom-up order and each node in the expression tree is annotated with bit sections of the expression's operands that contribute to the computation of bit sections of the intermediate value represented by the node. In our algorithm the intermediate value associated with an expression tree node during evaluation of the expression is denoted by *nval*. We also use the following basic notations:

- $var : \{[(l,h),s]\}$. Bit section value $nval_{l+s+1..h+s}$ is a function of bit section $var_{l+1..h}$. If s is 0, the bit sections of var and $nval$ refer to the same bit positions. Otherwise, a non-zero value of s indicates that the bit sections refer to different bit positions which is achieved by using left or right shift ($<<, >>$).
- $0/1 : \{[(l,h),s]\}$. Bit section value $nval_{l+s+1..h+s} = 0/1$, that is, we have a constant bit section with all bits being equal 0 or 1.

For the ease of presentation the following short hand notations are used as well:

- $var : \{[(l,m,h),s]\} \equiv var : \{[(l,m),s],[(m,h),s]\}$. Short hand notation for expressing adjacent bit sections of variable var.
- $var : \{[(l_1,h_1),s_1],[(l_2,h_2),s_2],...\}$ or $0/1 : \{[(l_1,h_1),s_1],[(l_2,h_2),s_2],...\}$. Short hand notation for multiple non-adjacent bit sections.

Finally, we introduce the following operations for value ranges. These operations are used in the computation of bit sections throughout the paper.

$$(l_1, h_1) \cap (l_2, h_2) = \begin{cases} (max(l_1, l_2), min(h_1, h_2)) & if\ max(l_1, l_2) < min(h_1, h_2) \\ \phi & otherwise \end{cases}$$

$$(l_1, h_1) \cup (l_2, h_2) = \begin{cases} \{(l_1, l_2), (l_2, h_2), (h_2, h_1)\} = (l_1, l_2, h_2, h_1) & if\ l_1 < l_2 < h_2 < h_1 \\ \{(l_1, l_2), (l_2, h_1), (h_1, h_2)\} = (l_1, l_2, h_1, h_2) & if\ l_1 < l_2 < h_1 < h_2 \\ \{(l_2, l_1), (l_1, h_2), (h_2, h_1)\} = (l_2, l_1, h_2, h_1) & if\ l_2 < l_1 < h_2 < h_1 \\ \{(l_2, l_1), (l_1, h_1), (h_1, h_2)\} = (l_2, l_1, h_1, h_2) & if\ l_2 < l_1 < h_1 < h_2 \\ \{(l_1, h_1), (l_2, h_2)\} & otherwise \end{cases}$$

$$(l_1, h_1) - (l_2, h_2) = \begin{cases} \{(l_1, l_2), (h_2, h_1)\} & if\ l_1 < l_2 < h_2 < h_1 \\ (l_1, l_2) & if\ l_1 < l_2 < h_1 < h_2 \\ (h_2, h_1) & if\ l_2 < l_1 < h_2 < h_1 \\ \phi & if\ l_2 < l_1 < h_1 < h_2 \\ (l_1, h_1) & otherwise \end{cases}$$

Algorithm. Visit the nodes in the expression tree in a bottom-up order applying steps 1 and 2 to them. Identify the bit sections of the lhs variable in step 3.

1. **Compute node annotations exploiting characteristics of operators and operands.**
   - *Variable leaf node.* Annotate node with $x : \{[(0, 32), 0]\}$, where variable x is the operand associated with the leaf node (and 32 is the bit width).
   - *Constant leaf node.* Annotate node with a set of bit sections each of which contains only 0's or 1's, that is, $0 : \{[(l0_1, h0_1), s0_1], [(l0_2, h0_2), s0_2], ...\}$ and $1 : \{[(l1_1, h1_1), s1_1], [(l1_2, h1_2), s1_2], ...\}$.
   - *Bitwise And* (&) *operator.* We exploit the following properties in computing the annotations for the *And* node: $a\&1 = a$, $a\&0 = 0$.

operand annotations	&'s annotation
$var : \{[(l_1, h_1), s_1]\}$,	$0 : \{[l_2, h_2], 0]\}$,
$0 : \{[(l_2, h_2), 0]\}$	$var : \{[(l' - s_1, h' - s_1), s_1]\}$ where
	$(l', h') = (l_1 + s_1, h_1 + s_1) - (l_1 + s_1, h_1 + s_1) \cap (l_2, h_2)$
$var : \{[(l_1, h_1), s_1]\}$,	$var : \{[(l_1, h_1), s_1]\}$,
$1 : \{[(l_2, h_2), 0]\}$	$1 : \{[(l_2, h_2) - (l_1 + s_1, h_1 + s_1) \cap (l_2, h_2), 0]\}$

   - *Bitwise Or* (\|) *operator.* We exploit the following properties in computing the annotations for the *Or* node: $a|1 = 1$, $a|0 = a$.

operand annotations	|'s annotation
$var : \{[(l_1, h_1), s_1]\}$,	$1 : \{[l_2, h_2], 0]\}$,
$1 : \{[(l_2, h_2), 0]\}$	$var : \{[(l' - s_1, h' - s_1), s_1]\}$ where
	$(l', h') = (l_1 + s_1, h_1 + s_1) - (l_1 + s_1, h_1 + s_1) \cap (l_2, h_2)$
$var : \{[(l_1, h_1), s_1]\}$,	$var : \{[(l_1, h_1), s_1]\}$,
$0 : \{[(l_2, h_2), 0]\}$	$0 : \{[(l_2, h_2) - (l_1 + s_1, h_1 + s_1) \cap (l_2, h_2), 0]\}$

- *Not operation (not nval).* Corresponding to constant bit sections in $nval$ create constant bit sections for the *not* node where 0 bit sections are converted into 1 bit sections and 1 bit sections are converted into 0 bit sections. That is, if $0/1 : \{[(l,h),0],..\}$ annotates $nval$, then $1/0 : \{[(l,h),0],..\}$ annotates *not*.
- *Constant left shift (nval << c where c is a constant ≤ 32).* From a bit section that is an annotation of $nval$, compute bit sections that annotate the << node as follows.

$nval$'s annotation	<<'s annotation
$var/1 : \{[(l,h),s]\}$	$0 : \{[(0,c),0]\}$ and $var/1 : \{[(l'-s-c, h'-s-c), s+c]\}$, where $(l',h') = (l+s+c, h+s+c) \cap (0,32)$
$0 : \{[(l,h),0]\}$	$0 : \{[(l',h'),0]\}$, where $(l',h') = (l+c, h+c) \cap (0,32)$ and $0 : \{[(0,c),0]\}$
$0 : \{[(0,h),0]\}$	$0 : \{[(0,h+c) \cap (0,32),0]\}$

- *Constant right shift (nval >> c).* From a bit section that is an annotation of $nval$, compute bit sections that annotate the >> node as follows. The following is applicable for shifting of an unsigned value. In case of a signed value, if the sign is known, similar rules can be derived.

$nval$'s annotation	>>'s annotation
$var/1 : \{[(l,h),s]\}$	$0 : \{[(32-c,32),0]\}$ and $var/1 : \{[(l'-s+c, h'-s+c), s-c]\}$, where $(l',h') = (l+s-c, h+s-c) \cap (0,32)$
$0 : \{[(l,h),0]\}$	$0 : \{[(32-c,32),0]\}$ and $0 : \{[(l',h'),0]\}$, where $(l',h') = (l-c, h-c) \cap (0,32)$
$0 : \{[(l,32),0]\}$	$0 : \{[(l-c,32) \cap (0,32),0]\}$

2. **Ensure all bits within a section are computed identically.** Closer examination of bit sections of different operand variables that annotate a given node can reveal whether further splitting of these bit sections is required to ensure that each resulting bit section is computed by exactly one expression. The bit section $var_1 : \{[(l_1,h_1),s_1]\}$ is split by bit section $var_2 : \{[(l_2,h_2),s_2]\}$, denoted by var_1/var_2, at a node in the expression tree by means of the following rule:

$$\frac{var_1 : \{[(l_1,h_1),s_1]\}}{var_2 : \{[(l_2,h_2),s_2]\}} = \begin{cases} var_1 : \{[(l_1, l_2+s_2-s_1, h_2+s_2-s_1, h_1), s_1]\} \\ \quad if\ l_1+s_1 < l_2+s_2 < h_2+s_2 < h_1+s_1 \\ var_1 : \{[(l_1, l2+s_2-s_1, h_1], s_1\} \\ \quad if\ l_1+s_1 < l_2+s_2 < h_1+s_1 < h_2+s_2 \\ var_1 : \{[(l_1, h_2+s_2-s_1, h_1), s_1]\} \\ \quad if\ l_2+s_2 < l_1+s_1 < h_2+s_2 < h_1+s_1 \\ var_1 : \{[(l_1,h_1),s_1]\} \\ \quad otherwise \end{cases}$$

The splitting is performed by considering every ordered pair of bit sections. As we can see, the above bit sectioning is performed to distinguish between bit sections which are computed differently by both bit sections var_1 and var_2. More precisely, we distinguish a bit section which is computed from both var_1 and var_2 from one which is computed only from var_1.

3. **Identify bit sections for the lhs variable.** After steps 1 and 2, the annotations of the root node of the expression tree are used to identify the bit sections of the variable on the left hand side. Let us assume that the width of a word is 32 bits, then we split the initial bit section of the lhs variable $var_{lhs} : \{[(0, 32), 0]\}$ if parts are computed differently. More formally, new bit sections are obtained by a repeated evaluation of

$$\frac{var_{lhs} : section}{any : \{[(l, h), s]\}}$$

for each annotation $any : \{[(l, h), s]\}$ at the root node of the rhs tree.

II. Generating bit section assignments. In this step we generate the bit section assignments corresponding to the bit sections identified for a lhs variable of a complex assignment. Given a bit section $v_{l+1..h}$, the expression which has to be assigned to $v_{l+1..h}$ is returned by the function call $genexp((l, h), e_{root})$, where e_{root} is the root node of the entire expression tree, i.e., for each bit section (l, h) for a lhs variable v we call

$$v_{l+1..h} := simplify(genexp((l, h), e_{root})).$$

Function *simplify* is the last step in which trivial patterns like "$a|0$" or "$a\&1$" are reduced to "a". As shown in Fig. 5, $genexp()$ traverses the expression examining the bit sections that annotate each node in order to find those that contribute to bits $l + 1..h$. If only one of the bit sections at a node contributes to bits $l + 1..h$, a traversal of the subtree is not required any more. In this case the operand is a sequence of $h-l$ bits belonging to a variable or it consists of constant (0 or 1) bits. If multiple bit sections contribute to bits $l + 1..h$, then the operator represented by the current node is included in the expression and the subexpressions that are its operands are identified by recursively applying $genexp()$ to the descendants.

Example. The example in Fig. 6 illustrates how the bit sections of lhs variable a of Fig. 1 are determined. The nodes are dealt with in a bottom-up manner. At the leave nodes, variables are initialized with whole bit sections, while constants are partitioned such that sequences of 0 or 1 are identified by bit sections. The annotations at the &-nodes indicate that some bits are 0 while others are derived from bit sections of variables a and b.

Now let us apply the $genexp()$ algorithm to the example. For bit sections $a_{1..4}$ and $a_{17..32}$ we find the contributing bit sections $0 : [(0, 4), 0]$ and $0 : [(16, 32), 0]$ which annotate the root node e_1 resulting in an assignment of constant 0. For both $a_{5..8}$ and $a_{13..16}$ we find a single contributing bit section that annotates e_1. From $b : [(0, 4), 4]$ we obtain that $a_{5..8}$ is assigned $b_{1..4}$ and from $a : [(12, 16), 0]$ we get that $a_{13..16}$ is assigned to itself, that is, it remains unchanged. Finally, for bit section $a_{9..12}$ we detect that there are two contributing bit sections, $a : [(8, 12), 0]$

```
genexp((l,h),e) {
  BS = φ
  for each section any : [(el,eh),es] ∈ set of annotations of node e do
     if range (l,h) is contained in range (el+es, eh+es) then
        BS = BS ∪ {any : [(el,eh),es]}
     endif
  endfor
  if BS == {any : [(el,eh),es]} then
     return ("any_{l-es+1..h-es}")
  else
    let e.lchild and e.rchild be expression trees for operands of e
    case e.op of
      e.op == "not" : return ("not" genexp((l,h),e.lchild);
      e.op == " << c" : return(genexp((l-c,h-c),e.lchild);
      e.op == " >> c" : return(genexp((l+c,h+c),e.lchild);
      e.op == "&" : return(genexp((l,h),e.lchild) "&" genexp((l,h),e.rchild);
      e.op == "|" : return(genexp((l,h),e.lchild) "|" genexp((l,h),e.rchild));
    end case
  endif
}
```

Fig. 5. Generating Bit Section Assignments.

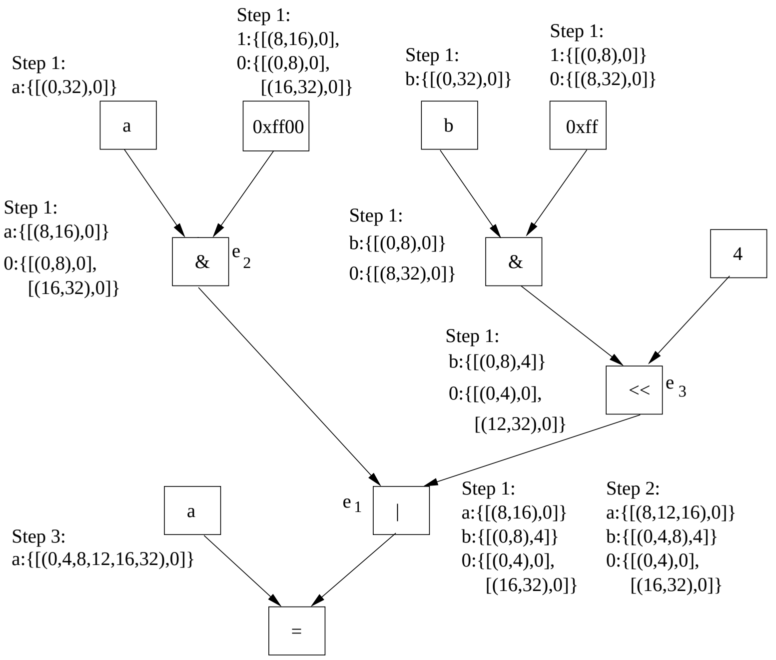


Fig. 6. Identifying Relevant Bit Sections for a Complex Assignment.

and $b : [(4,8),4]$. Therefore the operator $|$ at node e_1 is part of the expression and we must traverse the descendant nodes to locate the operands. In this case we find the operands $a_{9..12}$ and $b_{5..8}$ at the left and right nodes e_2 and e_3 respectively.

4 Global Phase: Placement of Split and Combine Nodes

In this phase global analysis is performed to relate the bit sections introduced in the local phase to each other by inserting split and combine nodes. Note that the analysis for insertion of splits and combines of one variable is independent of other variables.

Backward and forward propagation of bit sections. In order to determine whether an existing bit section should be eliminated using a *combine node* at a given program point, we must know whether the bit section is used later on in the program. This is accomplished by computing *anticipable bit section references* in a *backward analysis*. Similarly for determining whether a bit section should be created using a *split node* at a program point, we must know if the bit section already exists. This is accomplished by computing *available bit sections* in a *forward analysis*.

The values of data flow variables involved in this analysis are a set of bit sections belonging to a program variable. As before, each bit section is represented by a range (l, h) which denotes bits $l+1$ through h. We already defined and used operations $\cap$, $\cup$, and $-$ for value ranges. Since the data flow values are sets of value ranges, we define analogous operations over a set of value ranges whereby the new set is computed by considering every pair. We denote these operations by $\bigcap$, $\bigcup$, and $—$.

Given the above operations, the computation of anticipable (B) and available (F) bit sections for node n and variable v is shown below. The B and F sets are computed at the beginning (in) and end (out) of each node. $Ref[n, v]$ is a local set that represents the bit sections of v that are referenced on the lhs or rhs at node n.[3]

$$
\begin{array}{l}
\textit{Anticipable Bit Sections}: \ \textit{Backward Analysis} \\
\quad \textit{Initialize} \\
\qquad B_{out}[exit, v] \ = \ \phi \\
\quad \textit{Propagate} \\
\qquad B_{out}[n, v] \ = \ \bigcup_{s \ \in \ succ(n)} \ B_{in}[s, v] \\
\qquad B_{in}[n, v] \ = \ (B_{out}[n, v] \ — \ Ref[n, v]) \ \bigcup \ Ref[n, v]
\end{array}
$$

$$
\begin{array}{l}
\textit{Available Bit Sections}: \ \textit{Forward Analysis} \\
\quad \textit{Initialize} \\
\qquad F_{in}[entry, v] \ = \ \phi \\
\quad \textit{Propagate} \\
\qquad F_{in}[n, v] \ = \ \bigcup_{p \ \in \ pred(n)} \ F_{out}[p, v] \\
\qquad F_{out}[n, v] \ = \ (F_{in}[n, v] \ — \ Ref[n, v]) \ \bigcup \ Ref[n, v]
\end{array}
$$

[3] It should be noted that due to the semantics of the operators — and $\bigcup$, $(X — Ref) \bigcup Ref$ is NOT equal to $X \bigcup Ref$.

Combine and Split Node Insertion. The insertion points are determined based on the results of the forward and backward analysis. This is done in three steps. First, we identify all *candidate sections* at each program point that may be either *eliminated* or *created* at that point. We refer to these sections as *combine candidates* and *split candidates*, respectively. The combine candidate set, $CC_{in/out}(n,v)$, and the split candidate set $SC_{in/out}(n,v)$, for a program point n and variable v are identified as follows.

If a bit section of v is *available* at n's entry (exit) but *not anticipable* at n's entry (exit), then the *combine* node needed to eliminate the bit section is a legal candidate for insertion at entry (exit) of n. Similarly if a bit section of v is *anticipable* at n's entry (exit) but *not available* at n's entry (exit), then the *split* node needed to create the bit section is a legal candidate for insertion at entry (exit) of n. In the equations below $F^*_{in/out}$ and $B^*_{in/out}$ denote the solutions of the equation systems. Note that the only sections of interest are those that are smaller than $(0,32)$ and therefore $(0,32)$ is never included in the CC and SC sets. This is because creation of $(0,32)$ does not require a split as there is no larger section from which $(0,32)$ can be split and elimination of $(0,32)$ does not require a combine because there is no larger section into which $(0,32)$ can be merged.

Combine and Split Candidates

$$\begin{aligned}
CC_{in/out}(n,v) &= \{(l,h):\ (l,h)\in\ F^*_{in/out}(n,v)\ \textit{and}\ (l,h)\notin\ B^*_{in/out}(n,v)\} \\
&\quad - \{(0,32)\} \\
SC_{in/out}(n,v) &= \{(l,h):\ (l,h)\in\ B^*_{in/out}(n,v)\ \textit{and}\ (l,h)\notin\ F^*_{in/out}(n,v)\} \\
&\quad - \{(0,32)\}
\end{aligned}$$

In the second step we identify the *earliest points* for *combines* and *latest points* for *splits* for insertion of combine and split nodes. This can be done easily from the results of the first step by comparing the CC and SC sets of predecessor and successor nodes. For example, if a section is present in the CC_{in} set of a node, but not in any of the CC_{out} sets of its predecessor nodes, then the entry point of the node is the earliest point at which the combine can be placed.

Combine and Split Insertion Points

$$\begin{aligned}
EarliestCC_{in}(n,v) &= CC_{in}(n,v) - \textstyle\bigcup_{p\in\ pred(n)} CC_{out}(p,v) \\
EarliestCC_{out}(n,v) &= CC_{out}(n,v) - CC_{in}(n,v) \\
LatestSC_{in}(n,v) &= SC_{in}(n,v) - SC_{out}(n,v) \\
LatestSC_{out}(n,v) &= SC_{out}(n,v) - \textstyle\bigcup_{s\in\ succ(n)} SC_{in}(s,v)
\end{aligned}$$

Finally in the third step we *insert* combine and split nodes. To this end, during backward and forward analysis as well as during computation of combine and split candidates, the bit sections in the data flow sets are distinguished to fall into two categories: Those which are added to the sets because references to them are encountered and those which are added as a consequence of adding the ones which are referenced. Only the bit sections that are marked as being directly referenced are considered during insertion of split and combine nodes. The insertion conditions are given below. After the combine nodes the split nodes are inserted. The portions of the combine and split nodes which are not specified and marked by dots are determined by examining the remaining bit sections that exist at that program point.

Combine and Split Insertion
$\quad if\ (l, h)\ \in\ EarliestCC_{in/out}(n, v)\ then$
$\quad\quad insert\ combine\ node\ \text{'}v_{....}\ :=\ \oplus\ (...., v_{l+1..h}, ...)\text{'}$
$\quad if\ (l, h)\ \in\ LatestSC_{in/out}(n, v)\ then$
$\quad\quad insert\ split\ node\ \text{'}(...., v_{l+1..h}, ...)\ :=\ \oslash\ v_{....}\text{'}$

5 Concluding Remarks

We presented a novel program representation which supports analyses at bit section level. Instead of coping with subword accesses for each optimization, our representation makes those accesses explicit enabling to realize traditional analysis on subword level very easily. In a local analysis phase we analyze statement by statement and identify bit sections which could be of interest for subsequent optimization phases. Then we relate those bit sections to each other by introducing split and combine nodes.

References

1. T.M. Conte, P.K. Dubey, M.D. Jennings, R.B. Lee, A. Peleg, S. Rathnam, M. Schlansker, P. Song, and A. Wolfe, "Challenges of Combining General-Purpose and Multimedia Processors," *IEEE Computer*, Vol. 30, No. 12, pages 33–37, Dec. 1997.
2. J. Davidson and S. Jinturkar, "Memory access coalescing : a technique for eliminating redundant memory accesses," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 186–195, 1994.
3. J. Fridman, "Data Alignment for Sub-Word Parallelism in DSP," *IEEE Workshop on Signal Processing Systems* (SiPS), pages 251-260, 1999.
4. S. Larsen and S. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *ACM SIGPLAN Conf. on Programming Language Design and Implementation* (PLDI), pages 145–156, Vancouver B.C., Canada, June 2000.
5. X. Nie, L. Gazsi, F. Engel, and G. Fettweis, "A New Network Processor Architecture for High Speed Communications," *IEEE Workshop on Signal Processing Systems* (SiPS), pages 548-557, 1999.
6. A. Peleg and U. Weiser, "MMX Technology Extension to Intel Architecture," *IEEE Computer*, 16(4):42-50, August 1996.
7. M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," *ACM SIGPLAN Conf. on Programming Language Design and Implementation* (PLDI), pages 108–120, Vancouver B.C., Canada, June 2000.
8. J. Wagner and R. Leupers, "C Compiler Design for an Industrial Network Processor," *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (LCTES), pages 155-164, June 2001.
9. Y. Zhang and R. Gupta, "Data Compression Transformations for Dynamically Allocated Data Structures," *International Conference on Compiler Construction* (CC), Grenoble, France, April 2002.